**CS2106 Introduction to Operating Systems**

**Lab 3**

**Answer Book**

Please read the instructions in the main lab sheet before completing this document. Submission deadline is **1 pm, Sunday 31 March 2024**. The folder will stay open slightly after this, but once the folder closes, **absolutely no submissions will be allowed.**

**Submission checklist:** A ZIP file called AxxxxxxY.zip, where AxxxxxxY is the student ID of the student submitting. The ZIP file should contain:

* Your answer book, properly renamed.
* Your barrier.c and barrier.h
* Your sum-par.c

|  |  |
| --- | --- |
| **Student 1** | |
| Name: Eunice Koh Shu Ning |  |
| Student ID (A0256986E): |  |
| Group (B08): |  |
| **Student 2** | |
| Name: |  |
| Student ID (AxxxxxxY): |  |
| Group (Bxx): |  |

**Part 1**

Question 1.1 (1 mark)

This pattern indicates that the time quantum for each child process is such that it allows each process to complete its entire execution before the next one gets a chance to run. The time quantum is long enough for a process to complete its task before being preempted.

Question 1.2 (1 mark)

The final counter value is 0 because each child process operates independently of the others, and they each have their own copy of the counter variable. In the parent process, where the final counter value is printed, the counter variable has not been modified by any of the child processes.

Question 1.3 (1 mark)

The output suggests that the time quantum allows each process to execute once before the time quantum expires, preempting the next child process in a round-robin fashion.

Question 1.4 (1 mark)

// creates a shared memory object named "/counter\_memory". It opens the shared memory object for reading and writing

shm\_fd = shm\_open("/counter\_memory", O\_CREAT | O\_RDWR, 0666);

if (shm\_fd == -1) {

perror("shm\_open");

exit(EXIT\_FAILURE);

}

// If the creation fails, it prints an error message and exits the program

if (ftruncate(shm\_fd, SHARED\_MEMORY\_SIZE) == -1) {

perror("ftruncate");

exit(EXIT\_FAILURE);

}

// Map the shared memory

counter\_ptr = (int \*)mmap(NULL, SHARED\_MEMORY\_SIZE, PROT\_READ | PROT\_WRITE,

MAP\_SHARED, shm\_fd, 0);

if (counter\_ptr == MAP\_FAILED) {

perror("mmap");

exit(EXIT\_FAILURE);

}

// Unmap and close shared memory

if (munmap(counter\_ptr, SHARED\_MEMORY\_SIZE) == -1) {

perror("munmap");

exit(EXIT\_FAILURE);

}

close(shm\_fd);

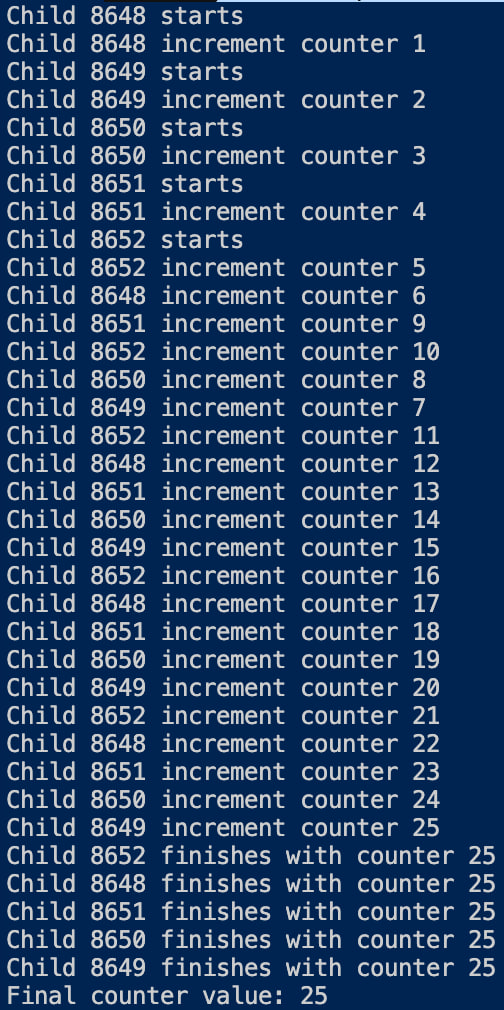
// Remove shared memory object

if (shm\_unlink("/counter\_memory") == -1) {

perror("shm\_unlink");

exit(EXIT\_FAILURE);

}



Question 1.5 (1 mark)

With more child processes trying to access and modify the shared counter variable, there's increased competition for this resource. This can lead to race conditions, where the final value of the counter may not be as expected due to unpredictable interleavings of increment operations by different processes.

Question 1.6 (1 mark)

Deadlocks can occur even in scenarios where locks are employed due to their lack of priority control. For instance, consider a situation where Child 3 requires a resource held by Child 2, but due to the absence of priority control, Child 2 is scheduled to acquire the lock last in the sequence, after Child 3. Child 3 may find itself unable to proceed because the resource it needs is held by Child 2, which is yet to acquire the lock. This situation can lead to a deadlock, where Child 3 is blocked, waiting for a resource held by Child 2.

Question 1.7 (1 mark)

Question 1.8 (1 mark)

Question 1.9 (1 mark)

Question 1.10 (1 mark)

**Part 2**

Question 2.1 (1 mark)

Question 2.2 (1 mark)

**Part 3**

Question 3.1 (1 mark)

(For grader only)

**Report: \_\_\_\_\_\_\_\_\_\_\_\_ / 13**

**Demo: \_\_\_\_\_\_\_\_\_\_\_\_\_ /7**

**TOTAL: \_\_\_\_\_\_\_\_\_\_\_\_\_/20**